Spark: High-speed distributed computing made easy with Spark

[1]Sangeeta M.Borde,[2]Kavita B.Waghmare,[3]Bareen k.Shaikh

Mit Arts, Commerce & Science College, Alandi (D), Pune

[1]smborde@mitacsc.ac.in,[2]kbwaghmare@mitacsc.ac.in,[3]bkshaikh@mitacsc.ac.in

Abstract

Resilient Distributed Datasets (RDDs) presents a distributed memory abstraction that allows programmers to perform in-memory computations on large clusters while retaining the fault tolerance of data flow models like MapReduce. RDDs are motivated by two types of applications that current data flow systems handle in-efficiently: iterative algorithms, which are common in graph applications and machine learning, and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitudes. To achieve fault tolerance efficiently, RDDs pro-vide a highly restricted form of shared memory: they are read-only datasets that can only be constructed through bulk operations on other RDDs. However, we show that RDDs are expressive enough to capture a wide class of computations, including MapReduce and specialized programming models for iterative jobs such as Pregel. Spark is a framework for writing fast, distributed programs. Spark solves similar problems as Hadoop MapReduce does but with a fast in-memory approach and a clean functional style API. Fast Data Processing with Spark covers how to write distributed MapReduce style programs with Spark.The primary reason to use Spark is for speed, and this comes from the fact that its execution can keep data in memory between stages rather than always persist back to HDFS after a Map or Reduce. Another reason to use Spark is its nicer high-level language compared to MapReduce. It provides a functional programming-like view that mimics Scala, which is far nicer than writing MapReduce code. (Although you have to either use Scala, or adopt the slightly-less-developed Java or Python APIs for Spark).

Keywords: RDDs (Resilient Distributed Datasets), MapReduce, API, Shark, Spark.

Introduction:

The past few years have seen a major change in computing systems, as growing data volumes require more and more applications to scale out to large clusters. In both commercial and scientific fields, new data sources and instruments (e.g., gene sequencers, RFID, and the web) are producing rapidly increasing amounts of information. Unfortunately, the processing and I/O capabilities of single machines have not kept up with this growth. As a result, more and more organizations have to scale out their computations across clusters.

The cluster environment comes with several challenges for programmability. The first one is parallelism: this setting requires rewriting applications in a parallel fashion, with programming models that can capture a wide range of computations. However, unlike in other parallel platforms, a second challenge in clusters is faults: both outright node failures and stragglers (slow nodes) become common at scale, and can greatly impact the performance of an application. Finally, clusters are typically shared between multiple users, requiring runtimes that can dynamically scale computations up and down and exacerbating the possibility of interference.

As a result, a wide range of new programming models have been designed for clusters. At first, Google's MapReduce presented a simple and general model for batch processing that automatically handles faults. However, MapReduce was found poorly suited for other types of workloads, leading to a wide range of specialized models that differed significantly from MapReduce. We can design a unified programming abstraction that not only captures these disparate workloads, but enables new applications as well. In particular, we show that a simple extension to MapReduce called Resilient Distributed Datasets (RDDs), which just adds efficient data sharing primitives, greatly increases its generality. The resulting architecture has several key advantages over current systems:

1. It supports batch, interactive, iterative and streaming computations in the same runtime enabling applications that combines these models and offering significantly higher performance for these combined applications than desperate systems.

2. It provides fault and straggler tolerance across these computation modes at a very low cost. Indeed, in several areas (streaming and SQL), our RDD-based approach yields new system designs with significantly stronger fault tolerance properties than the status quo.

3.It achieves performance that is often 100× better than MapReduce and comparable with specialized systems in individual application domains.

4. It is highly amenable to multitenancy, allowing applications to scale up and down elastically and share resources in a responsive fashion. We implement the RDD architecture in a stack of open source systems including Apache Spark as the common foundation; Shark for SQL processing; and Spark Streaming for distributed stream processing (Figure 1). We evaluate these systems using both real user applications and traditional benchmarks. Our implementations provide state-of-the-art performance for both traditional and new data analysis workloads, as well as the first platform to let users compose these workloads.

We implement the RDD architecture in a stack of open source systems including Apache Spark as the common foundation; Shark for SQL processing; and Spark Streaming for distributed stream processing (Figure 1.1)

I. Problems with Specialized Systems

Today's cluster computing systems are increasingly specialized for particular application domains. While models like MapReduce and Dryad [36, 61] aimed to capture fairly general computations, researchers and practitioners have since developed more and more specialized systems for new application domains. Recent

Figure 1. Computing stack implemented in this dissertation. Using the Spark implementation of RDDs, we build other processing models, such as streaming, SQLand graph computations, all of which can be intermixed within Spark programs. RDDs themselves execute applications as a series of fine-grained tasks, enabling efficient resource sharing.

Although specialized systems seem in natural way to scope down the challenges in distributed environment, they also come with several drawbacks--

1. Work duplication: Many specialized systems still need to solve the same underlying problems, such as work distribution and fault tolerance. For example, a distributed SQL engine or machine learning engine both need to perform parallel aggregations. With separate systems, these problems need tobe addressed anew for each domain.

2. Composition: It is both expensive and unwieldy to compose computations in different systems. For "big data" applications in particular, intermediate datasets are large and expensive to move. Current environments require exporting data to a replicated, stable storage system to share it between computing engines, which is often many times more expensive than the actual computations. Therefore, pipelines composed of multiple systems are often highly inefficient compared to a unified system.

3. Limited scope: If an application does not fit the programming model of a specialized system, the user must either modify it to make it fit within current systems, or else write a new runtime system for it.

4. Resource sharing: Dynamically sharing resources between computing engine is become difficult because most engines assume they own same set of machines for the duration of the application.

5. Management and administration: Separate systems require significantly more work to manage and deploy than a single one. Even for users, they require learning multiple APIs and execution models .Because of these limitations; a unified abstraction for cluster computing would have significant benefits in not only usability but also performance, especially for complex applications and multi-user settings

II.Resilient distributed datasets (RDDs)

To address this problem, we introduce a new abstraction, resilient distributed datasets (RDDs), that forms a simple extension to the MapReduce model. The insight behind RDDs is that although the workloads that MapReduce was unsuited for (e.g., iterative, interactive and streaming queries) seem at first very different, they all require a common feature that MapReduce lacks: efficient data sharing across parallel computation stages. With an efficient data sharing abstraction and Map Reduce like operators, all of this workload can be expressed efficiently, capturing the key optimizations in current specialized systems. RDDs offer such an abstraction for a broad set of parallel computations, in a manner that is both efficient and fault-tolerant. In particular, previous fault-tolerant processing models for clusters, such as MapReduce and Dryad, structured computations as a directed acyclic graph (DAG) of tasks.

Goals and Overview:

Our goal is to provide an abstraction that supports applications with working sets (i.e., applications that reuse an intermediate result in multiple parallel operations) while preserving the attractive properties of Map Reduce and related models: automatic fault tolerance, locality-aware scheduling, and scalability. RDDs should be as easy to program against as data flow models, but capable of efficiently expressing computations with working sets.

Out of our desired properties, the most difficult one to support efficiently is fault tolerance. In general, there are two options to make a distributed dataset fault-tolerant: check pointing the data or logging the updates made to it. In our target environment (large-scale data analytics), check pointing the data is expensive: it would require replicating big datasets across machines over the data-center network, which typically has much lower band-width than the memory bandwidth within a machine and it would also consume additional storage (replicating data in RAM would reduce the total amount that can be cached, while logging it to disk would slow down applications). Consequently, we choose to log up-dates. However, logging updates it also expensive if there are many of them. Consequently, RDDs only support coarse-grained transformations, where we can log a single operation to be applied to many records. We then remember the series of transformations used to build an RDD (i.e., its lineage) and use it to recover lost partitions.

III. Why a New Programming Model?

MapReduce greatly simplified big data analysis But as soon as it got popular, users wanted more:

1. More complex, multi-pass analytics (e.g. ML, graph)
2. More interactive ad-hoc queries
3. More real-time stream processing

All 3 need faster data sharing across parallel jobs.

Data Sharing in MapReduce

HDFS
Read

HDFS
Write

HDFS
Read

HDFS
Write

Input

HDFS Read

Input

Fig.2 Slow Due to Replication, Serilization & Disk I/O

Data Sharing in Spark

Input

One Time Processing

Input **Distributed Memory**

Fig.3 Data Sharing in Spark

MapReduce is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster.

A MapReduce program is composed of a **Map()** procedure that performs filtering and sorting (such as sorting students by first name into queues, one queue for each name) and a **Reduce()** procedure that performs a summary operation (such as counting the number of students in each queue, yielding name frequencies). The "MapReduce System" (also called "infrastructure" or "framework") orchestrates the processing by marshalling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy and fault tolerance.

The model is inspired by the map and reduce functions commonly used in functional programming, although their purpose in the MapReduce framework is not the same as in their original forms. The key contributions of the MapReduce framework are not the actual map and reduce functions, but the scalability and fault-tolerance achieved for a variety of applications by optimizing the execution engine once. As such, a single-threaded implementation of MapReduce (such as MongoDB) will usually not be faster than a traditional (non-MapReduce) implementation; any gains are usually only seen with multi-threaded implementations. Only when the optimized distributed shuffle operation (which reduces network communication cost) and fault tolerance features of the MapReduce framework come into play, is the use of this model beneficial.

MapReduce libraries have been written in many programming languages, with different levels of optimization. A popular open-source implementation is Apache Hadoop. The name MapReduce originally referred to the proprietary Google technology, but has since been generalized.

IV. What is Shark?

Shark is a large-scale data warehouse system that runs on top of Spark and is backward-compatible with Apache Hive, allowing users to run unmodified Hive queries on existing Hive workhouses. Shark is able to run Hive queries 100 times faster when the data fits in memory and up to 5-10 times faster when the data is stored on disk. Shark is a port of Apache Hive onto Spark that is compatible with existing Hive warehouses and queries. Shark can answer Hive SQL queries up to 100 times faster than Hive without modification to the data and queries, and is also open source as part of BDAS.This high-speed query engine runs Hive SQL queries on top of Spark up to 100x faster than Hive, and supports fault recovery and complex analytics (e.g. machine learning).

Spark:

Spark is a high-speed cluster computing system compatible with Hadoop that can outperform it by up to 100 times considering its ability to perform computations in memory. It is a computation engine built on top of the Hadoop Distributed File System (HDFS) that efficiently support iterative processing (e.g., ML algorithms), and interactive queries. Fast and expressive cluster computing system compatible with Apache Hadoop Improves efficiency through:

1. General execution graphs
2. In-memory storage

Improves usability through:

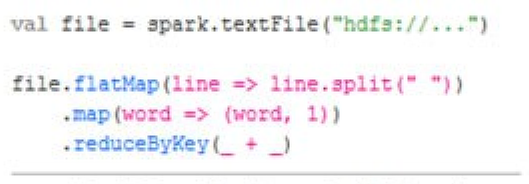
- Rich APIs in Scala, Java, Python
- Interactive shell

Spark is a framework for writing fast, distributed programs. Spark solves similar problems as Hadoop MapReduce does but with a fast in-memory approach and a clean functional style API. With its ability to integrate with Hadoop and inbuilt tools for interactive query analysis (Shark), large-scale graph processing and analysis (Bagel), and real-time analysis (Spark Streaming), it can be interactively used to quickly process and query Big data sets.

Fast Data Processing with Spark covers how to write distributed map reduce style programs with Spark. Spark provides an efficient abstraction for in-memory cluster computing called Resilient Distributed Datasets, and can run 100x faster than Hadoop for these applications.

Spark Architecture

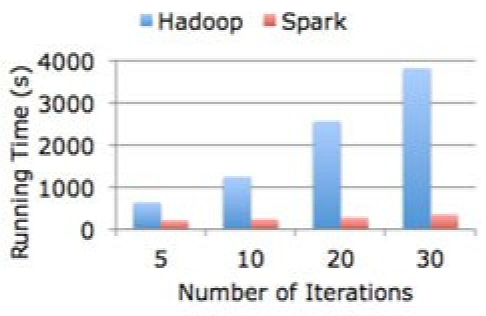
Spark is an open source cluster computing system developed in the UC Berkeley AMP Lab. The system aims to provide fast computations, fast writes, and highly interactive queries. Spark significantly outperforms Hadoop MapReduce for certain problem classes and provides a simple Ruby-like interpreter interface. See Figure 1(right) for examples.



```
val file = spark.textFile("hdfs://...")

file.flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey(_ + _)
```

Word Count implemented in Spark

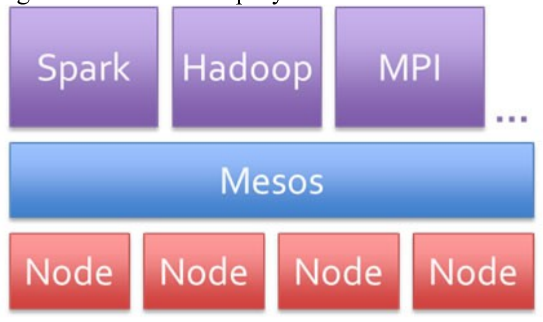


Logistic regression in Spark vs Hadoop

Spark beats Hadoop by providing primitives for in-memory cluster computing; thereby avoiding the I/O bottleneck between the individual jobs of an iterative MapReduce workflow that repeatedly performs computations on the same working set. Spark makes use of the Scala language, which allows distributed datasets to be manipulated like local collections and provides fast development and testing through it's interactive interpreter (similar to Python or Ruby).

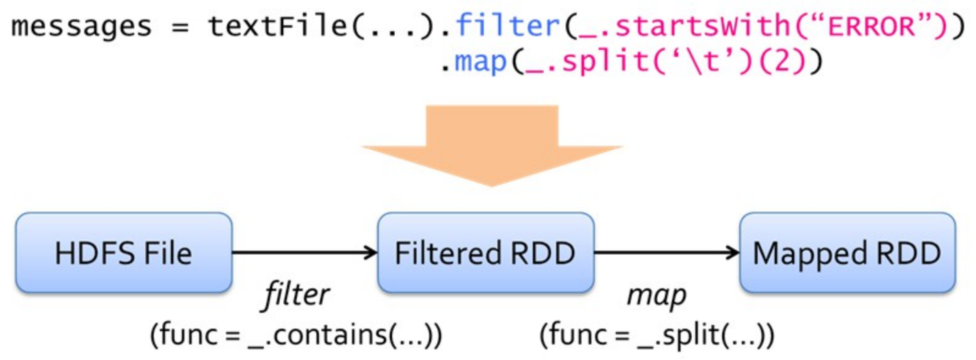
Spark was also developed to support interactive data mining, (in addition to the obvious utility for iterative algorithms). The system also has applicability for many general computing problems. Their site includes example programs for File Search, In-memory Search, and Estimating Pi, in addition to the two examples in Figure 1. Beyond these specifics Spark is well suited for most dataset transformation operations and computations.

Additionally, Spark is built to run on top of the Apache Mesos cluster manager. This allows Spark to operate on a cluster side-by-side with Hadoop, Message Passing Interface (MPI), Hyper Table, and other applications. This allows an organization to develop hybrid workflows that can benefit from both dataflow models, with the cost, management, and interoperability concerns that would arise from using two independent clusters.



As "they" often say, there is no free lunch! How does Spark provide its speed and avoid Disk I/O while retaining the attractive fault-tolerance, locality, and scalability properties of MapReduce? The answer from the Spark team is a memory abstraction they call a Resilient Distributed Dataset (RDD).

Existing distributed memory abstractions such as key-value stores and databases allow fine-grained updates to mutable state. This forces the cluster to replicate data or log updates across machines to maintain fault tolerance. Both of these approaches incur substantial overhead for a data-intensive workload. RDDs instead have a restricted interface that only allows coarse-grained updates that apply the same operation to many data items (such as map, filter, or join).

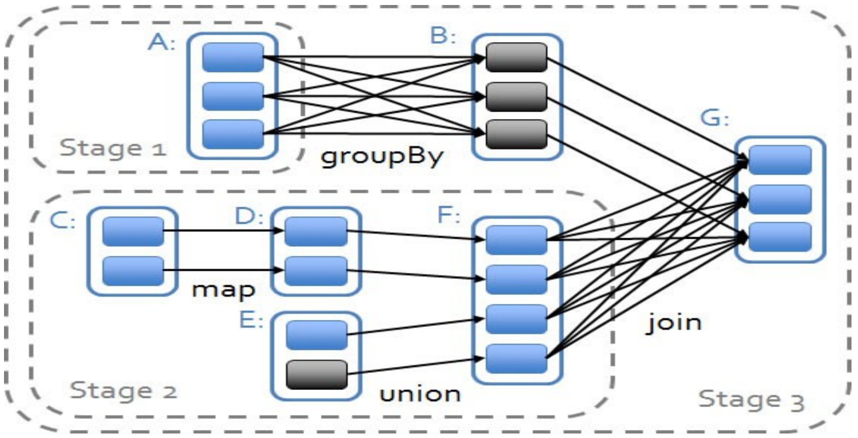


```
messages = textFile(...).filter(_.startsWith("ERROR"))
                        .map(_.split('\t')(2))
```

This allows Spark to provide fault-tolerance through logs that simply record the transformations used to build a dataset instead of all the data. This high-level log is called a lineage and the figure above shows a code snippet of the lineage being utilized. Since parallel applications, by their very nature, typically apply the same operations to a large portion of a dataset, the coarse-grained restriction is not as limiting as it might seem. In fact, the Spark paper showed that RDDs can efficiently express programming models from numerous separate frameworks including MapReduce, DryadLINQ, SQL, Pregel, and Hadoop.

Additionally, Spark also provides additional fault tolerance by allowing a user to specify a persistence condition on any transformation which causes it to immediately write to disk. Data locality is maintained by allowing users to control data partitioning based on a key in each record. (One example use of this partitioning scheme is to ensure that two datasets that will be joined are hashed in the same way.) Spark maintains scalability beyond memory limitations for scan-based operations by storing partitions that grow too large on disk.

As stated previously, Spark is primarily designed to support batch transformations. Any system that needs to make asynchronous fine grain updates to shared state (such as a data store or a web application) should use a more traditional system such as a database, RAM Cloud, or Piccolo.



Advantages of Spark----

Speed

Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk. Spark has an advanced DAG execution engine that supports cyclic data flow and in-memory computing.

Ease of Use

Write applications quickly in Java, Scala or Python.Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it *interactively* from the Scala and Python shells.

Generality

Combine SQL, streaming, and complex analytics. Spark powers a stack of high-level tools including Spark SQL, MLlib for machine learning, GraphX, and Spark Streaming. You can combine these frameworks seamlessly in the same application.

Integrated with Hadoop

Spark can run on Hadoop 2's YARN cluster manager, and can read any existing Hadoop data.If you have a Hadoop 2 cluster, you can run Spark without any installation needed. Otherwise, Spark is easy to run stand alone or on EC2 or Mesos. It can read from HDFS, HBase, Cassandra, and any Hadoop data source.

Conclusion:

The interesting fact about Spark is the approach to fault tolerance. Instead of persisting or check pointing intermediate results, Spark remembers the sequence of operations which led to a certain data set. So when a node fails, Spark reconstructs the data set based on the stored information. They argue that this is actually not that bad because the other nodes will help in the reconstruction.

Spark is in-memory by default, which naturally leads to a large improvement in performance, and even allows to run iterative algorithms. Spark has no built- in support for iterations, though, it's just that they claim it's so fast that you can run iterations if you want to. Hadoop in comparison seemed to be so much more, a distributed file system, obviously map reduce, with support for all kinds of data formats, data sources, unit testing, clustering variants, and so on and so on.

Spark also provides more complex operations like joins, group-by, or reduce-by operations so that you can model quite complex data flows (without iterations, though).Over time it dawned that the perceived simplicity of Spark actually said a lot more about the Java API of Hadoop than Spark. Even simple examples in Hadoop usually come with a lot of boilerplate code. Spark actually provides a non-trivial set of operations (really hard to tell just from the ubiquitous word count example), RDDs are the basic building block of Spark and are actually really something like distributed immutable collections. These define operations like map or for each which are easily parallelized, but also join operations which take two RDDs and collects entries based on a common key, as well as reduce-by operations which aggregates entries using a user specified function based on a given key. RDDs can be read from disk but are then held in memory for improved speed where they can also be cached so you don't have to reread them every time. That alone adds a lot of speed compared to Hadoop which is mostly disk based.

References:

- H. Li, A. Ghodsi, M. Zaharia, S. Shenker and I. Stoica, Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks, to appear at SOCC 2014.
- S.N. Naccache, S. Federman, N. Veeeraraghavan, M. Zaharia, D. Lee, E. Samayoa, J. Bouquet, A.L. Greninger, K. Luk, B. Enge, D.A. Wadford, S.L. Messenger, G.L. Genrich, K. Pellegrino, G. Grard, E. Leroy, B.S. Schneider, J.N. Fair, M.A. Martinez, P. Isa, J.A. Crump, J.L. DeRisi, T. Sittler, J. Hackett Jr., S. Miller and C.Y. Chiu, A Cloud-Compatible Bioinformatics Pipeline for Ultrarapid Pathogen Identification from Next-Generation Sequencing of Clinical Samples, Genome Research, June 2014.
- Spark Streaming AMP Camp Video: YouTube
- Spark & Spark Streaming Papers and Slides: html
- Spark Streaming - Google Groups: html
- Spark. html
- Hadoop for Real-Time(telruptive.com): htm
- Shivnath Babu. Towards automatic optimization of MapReduce programs.
- In SoCC'10, 2010.Azza Abouzeid et al. HadoopDB: an architectural hybrid of MapReduce and
- DBMS technologies for analytical workloads. VLDB, 2009.
- [2] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman,

- Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle.
- MillWheel: Fault-tolerant stream processing at internet scale. In VLDB, 2013.
-